

周报

本周的主要工作在教材的修改上。在参考了大量 BSP 和 OBB 教程，以及相关代码后，实现了基于 Solid Bsp Tree 的碰撞检测和基于 OBB 的碰撞检测游戏场景，如图 1,2,3,4,5 所示。图 1、2 和 3 是基于 BSP 树的碰撞检测游戏场景，一个简单的第一视角漫游游戏，同时可控制绿色球作为 AI 或者主角进行物体间的碰撞检测测试。

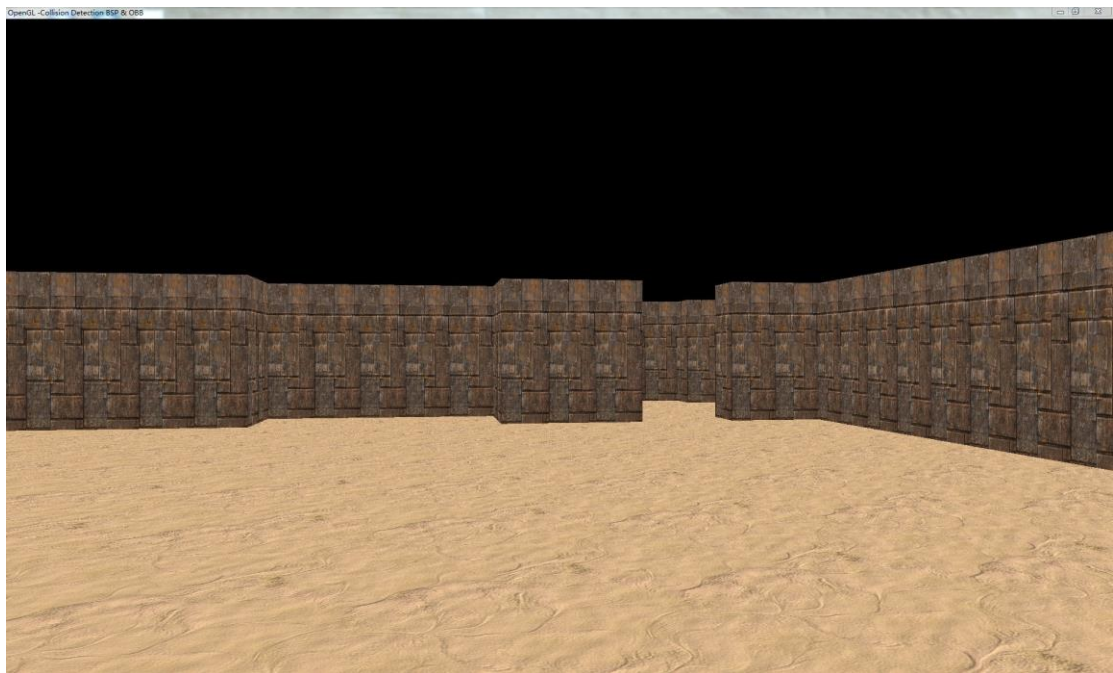


图 1 基于 BSP 树的碰撞检测游戏场景 1

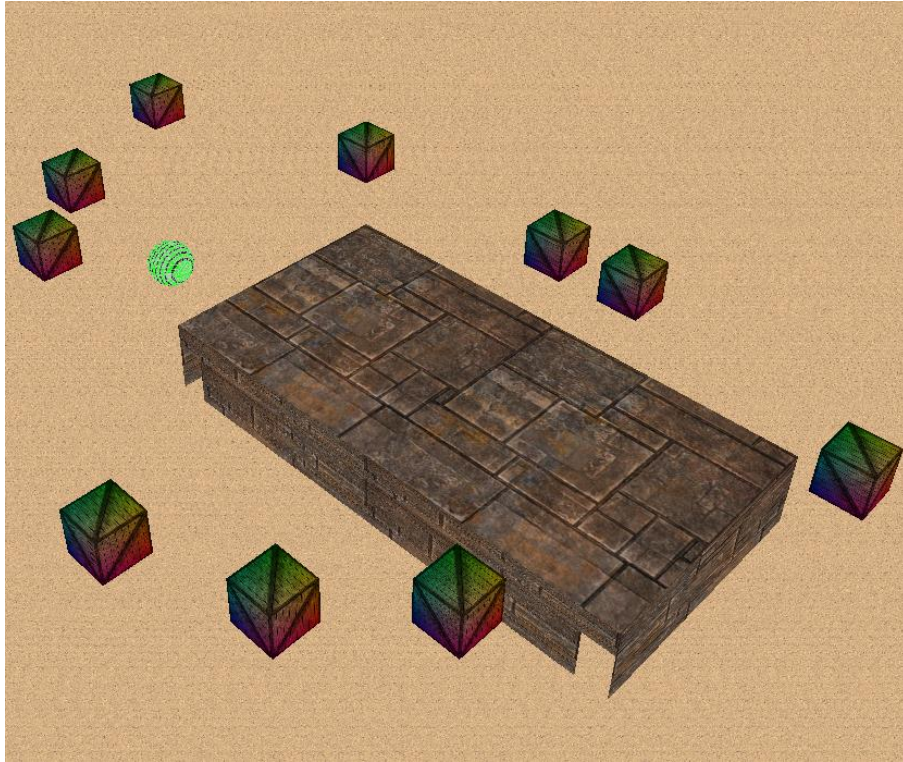


图 2 基于 BSP 树的碰撞检测游戏场景 2

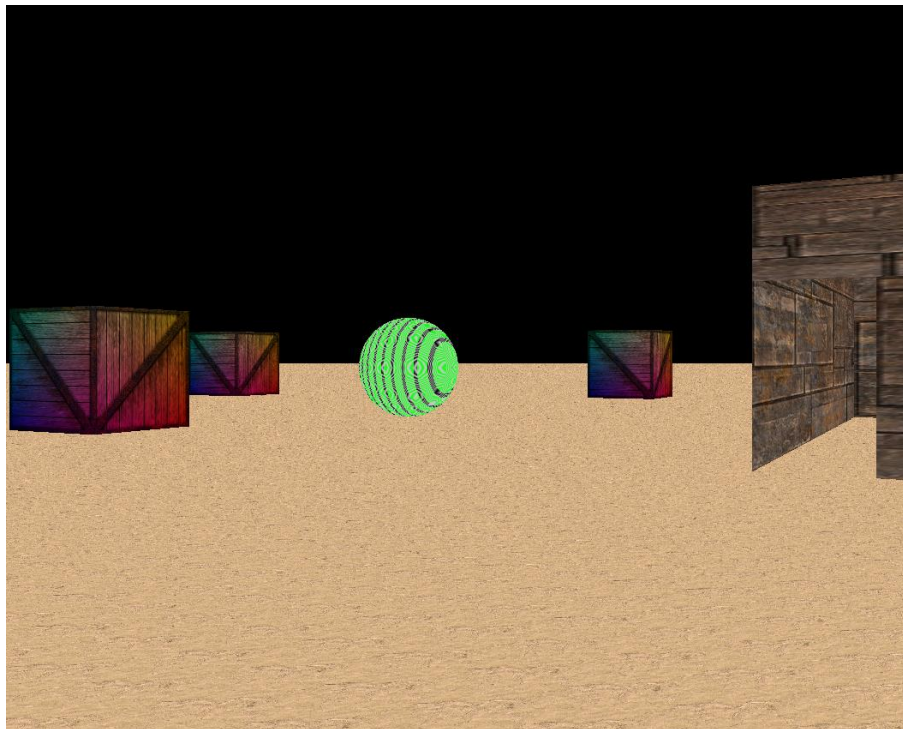


图 3 基于 BSP 树的碰撞检测游戏场景 3

图 4 和图 5 是基于 OBB 树的简单碰撞检测场景

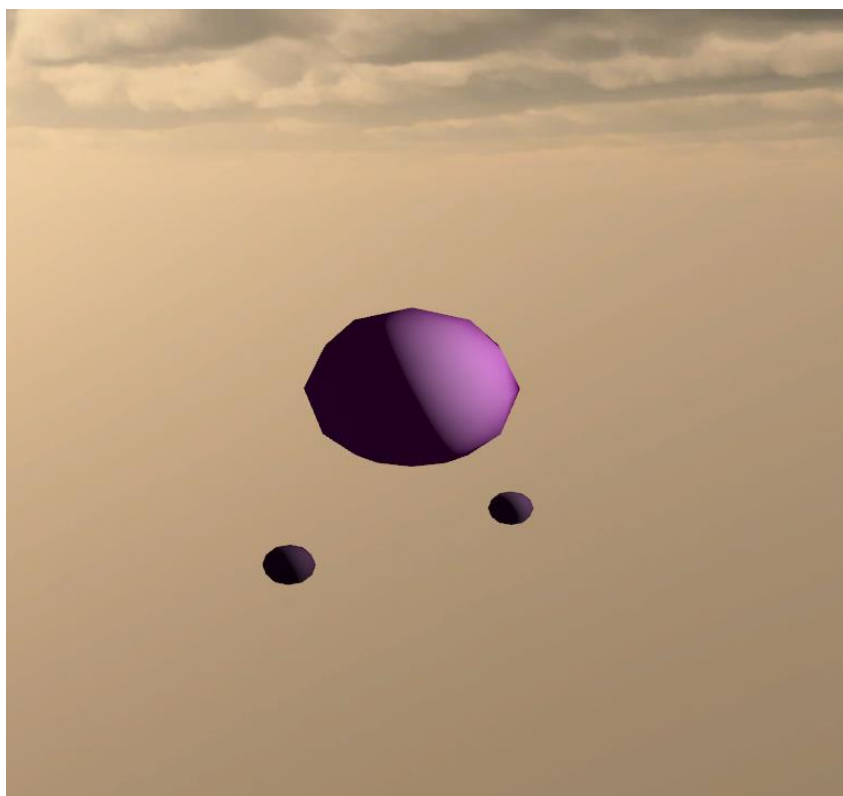


图 4 基于 OBB 树的简单碰撞检测场景：原始情况



图 5 基于 OBB 树的简单碰撞检测场景，碰撞后

关于 BSP 碰撞检测的伪代码如下

BSP 碰撞检测实例应用伪代码

```
VERTEX { // 顶点结构
float x, y, z; // 位置
float r, g, b, a; // 颜色
float tu, tv; // 纹理坐标
};

POLYGON { // 多边形结构
    VERTEX    VertexList[10];    // 构成多边形的顶点
    int        Indices[30];        // 索引值
        int    nNumOfVertices; // 构成多边形的顶点个数
        int    nNumOfIndices;  // 构成多边形的顶点的索引个数
    vector3    Normal;            // 多边形法向
        POLYGON *Next;            // 指向下一多边形的指针
};

NODE { // BSP 树节点结构
    POLYGON *Splitter; // 当前节点的分割平面
    NODE    *Front;     // 前向面节点指针
    NODE    *Back;      // 后向面节点指针
    bool     bIsLeaf;   // 叶子节点标志位，没有多边形
    bool     bIsSolid;  // 实心标志位；叶子节点；若指向父节点的前向面，值为 false；若指向父节点的背面，值为 true
};

POLYGON *g_polyList;    // 场景多边形链表
NODE     *g_RootBSPTree; // BSP 树
vector3    g_CameraPos; // 移动以后新的相机位置
vector3    g_PreCameraPos; // 移动之前的相机位置

Init() { // 初始化阶段
    LoadTextures(); // 载入纹理
    InitScenePolys(); // 构造场景多边形；载入场景模型，以三角形面片为单位加入到 g_polyList 中，构建场景多边形链表
    ConstructBspTree( g_RootBSPTree, g_polyList ); // 以场景多边形链表构建 BSP 树；以 g_RootBSPTree 为根节点，选取 g_polyList 中的一个平面为分割面，递归构建 BSP 树
}

Render() { // 绘制阶段
    CollisionDetection( g_CameraPos, g_PreCameraPos, g_RootBSPTree ); // 判断新旧相机位置与 BSP 树中分割面的位置关系，确保相机位置不发生穿越，保持正确的碰撞检测效果
    WalkBspTree( g_RootBSPTree, &g_CameraPos ); // 遍历、渲染 BSP 树中所有的多边形，绘制场景
}

bool CollisionDetection ( vector3*Start, vector3*End, NODE *Node ){
    int PointA = JudgeByDotProduct(Start, Node->Splitter);
    int PointB = JudgeByDotProduct (End, Node->Splitter);
    if(PointA 与 PointB 都在分割面上) {
return CollisionDetection (Start,End,Node->Front);}
    if( PointA 与 PointB 在分割面的两侧) {
```

```

        GetIntersect( Start, End, Node->Splitter->Normal, intersection); //计算新旧相机位置连线与分割面的
        交点，交点位置保存在 intersection 里
        return CollisionDetection ( Start, intersection, Node->Front) && CollisionDetection ( End, intersection,
        Node->Back);
    }
    if( PointA 或者 PointB 在分割面前向面) {
    return CollisionDetection ( Start, End, Node->Front );}
    else {
    return CollisionDetection ( Start, End, Node->Back );}
    return true;
}
void WalkBspTree( NODE *Node, vector3 *pos ){
    if(pos 在当前分割面前){
        if(当前节点 Node 的后向子节点不为空){
            WalkBspTree( Node->Back, pos );
        }
        绘制当前节点的分割面三角形;
        If(当前节点 Node 的前向子节点不为空){
            WalkBspTree( Node->Front, pos );
        }
    }
    If(当前节点 Node 的前向子节点不为空){
        WalkBspTree( Node->Front, pos );}
    If(当前节点 Node 的后向子节点不为空){
        WalkBspTree( Node->Back, pos );}
}

```

图 6 是 CollisionDetection 函数的主要过程图解。假设当前游戏场景由四面墙构成，箭头所指为外向面，四个面围成的内区域为墙体。四面墙的标号 A，B，C，D 分别按其在 BSP 树中出现的顺序先后给定。A 是根节点。图中两点 S 和 E 分别代表相机或者待检测物体的移动后位置和移动前位置，对应伪代码函数 CollisionDetection() 中的 Start 和 End。

首先，以根节点和新旧位置为参数调用 CollisionDetection 函数进行检测。如果两个点都在当前节点的前面或者后面，则分别递归地以新旧位置以及当前节点的前子节点和后子节点为参数调用 CollisionDetection 函数。若两个点都在当前节点所在面上，则同都在当前节点前面的情况。

若两个节点分别在当前节点所在面的两侧，如图所示，则进行如下处理。首先第一步调用可知 S 和 E 都在 A 的背面，以两点和 A 的后向面子节点指针（即 B）为参数调用 CollisionDetection 函数，同样，

可知两点都在 B 的背面，再以两点和 B 的后向面子节点指针（即 C）为参数调用 CollisionDetection 函数。此时 S 和 E 分别在 C 的两侧，于是计算 SE 与 C 的交点 CE，然后以 S、CE、C 的后向面子节点指针（即 D）以及 CE、E、C 的前向面子节点指针（即空指针 NULL）分别递归调用 CollisionDetection 函数。CE-E-NULL 的碰撞检测调用中可知此段路径中不存在碰撞。S-CE-D 的碰撞检测则更为复杂，因为在 S-CE 与 D 有交点。因此，再次计算该交点 CS。接着，以 S、CS、D 的前向面子节点指针（即空指针 NULL）以及 CS、CE、D 的后向面子节点指针（即空指针 NULL）分别递归调用 CollisionDetection 函数。可知 S-CS 路径不存在碰撞，而 CS-CE 路径都在墙体内，发生碰撞。于是最后的结果是 SE 之间发生碰撞，亦即相机或者待检测物体无法由当前路径在下一帧移动。

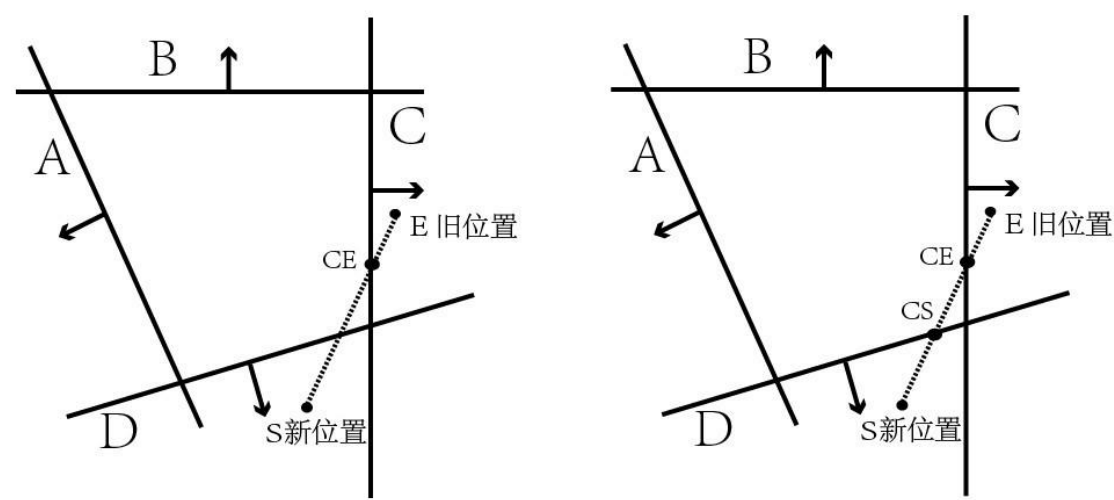


图 6CollisionDetection 函数过程图解

图 7 是基于 OBB 树碰撞检测的基本流程

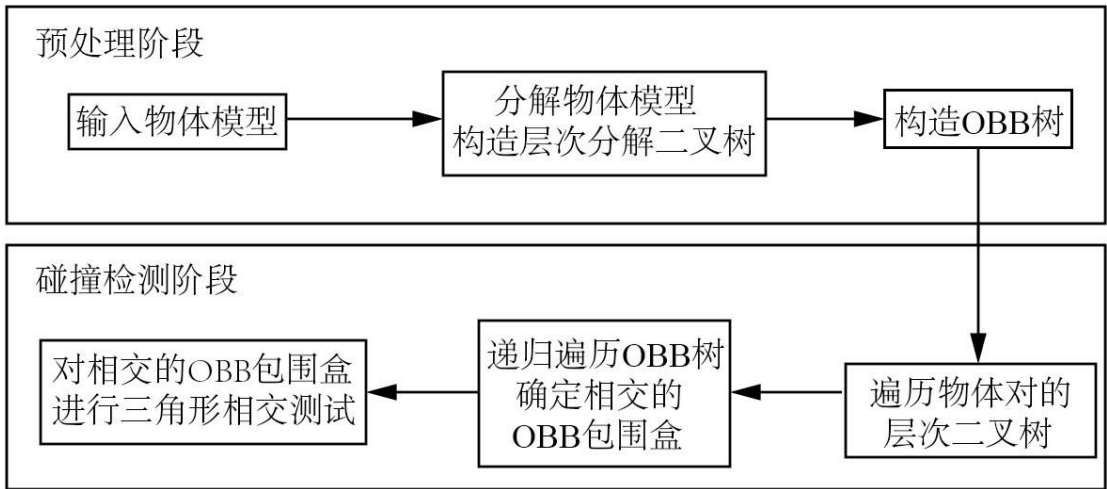


图 7 基于 OBB 树碰撞检测的基本流程

传统的基于 **OBB** 树的碰撞检测算法一般可分为两个阶段，一是预处理阶段，另一个是碰撞检测阶段。预处理阶段就是把物体进行分解，为分解得到的每个层次的物体块构造相应的 **OBB** 包围盒。碰撞检测运行阶段，遍历物体对的层次二叉树，层次树中的每一个节点与 **OBB** 包围的分解块相对应。遍历层次树时，先检测分解块的节点所对应的 **OBB** 包围盒是否相交；当两 **OBB** 包围盒相交时，对两 **OBB** 包围盒进行三角形的相交测试，得出碰撞检测结果。在进行 **OBB** 碰撞检测时，又分为两步，**OBB** 间的重叠测试和基本几何元素之间的相交测试。使用 **OBB** 树进行碰撞检测，其目的是通过两个对象的包围盒树中各节点所对应的 **OBB** 包围盒之间的重叠测试，尽可能早地排除不可能相交的基本几何元素对，仅对有可能相交的基本几何元素对进行精确的相交检测。**OBB** 包围盒间的重叠测试即可通过分离轴定理计算，而基本几何元素的相交测试则可通过对应的求交算法计算。

教材中新加入的内容大概如上所示，代码量和工作量还是很大，所以没能按预想的快点搞定，**SimHash** 的工作也没有开始。我在网上找到了旧版本教材的电子版，查阅了和前述章节重复的地方，主要是不同的物体包围盒示意那段，已经删除。**OBB** 的伪代码还没来得及写完，写完后再把文字润色下，预计再有一天即提交修改后的初稿。

此外，上周还完成了 **VisPaper Collection** 的最后一次大更新，包括 **EVIS**, **PVIS** 也进行了最后一次大更新。基本上不会有大的改动，等小

马上传服务器即可。

建刚因为没有拿到工作 OFFER，所以压力非常大，为了让他安心找工作，所以上周没让他来实验室，让他集中精力先去找工作。

我在完成教材的事情后，即可开始 SimHash 设想的初步实现。